Project Report

The problem:

As cameras become more prevalent in our daily lives, such as mobile devices and still cameras, addressing the widespread challenge of image blur becomes crucial to ensuring the effectiveness of various applications. In the world of photos, various factors contribute to the deterioration of image quality, including shaking hands during photography, limitations on the camera's autofocus capabilities, motion blur and adverse weather conditions. Moreover, insufficient lighting conditions, which is Common in real-world settings, the problem is exacerbated by reducing image sharpness and introducing noise. This degradation goes beyond visual aesthetics because the difficulty of distinguishing important details in images can lead to misinterpretations, putting the reliability of automatic recognition algorithms at risk. These real-world challenges therefore require a solution that can improve image clarity regardless of camera type or environmental conditions.

The Project Goals:

- Address Widespread Image Blur: Develop techniques to mitigate image blur caused by factors such as shaky hands, autofocus limitations, motion blur, adverse weather conditions, and insufficient lighting.
- Enhance Image Visibility in Real-World Scenarios: Focus on improving the clarity of images, especially in street scenes, where capturing clear signals is crucial. Real-world scenarios pose diverse challenges, and the project aims to navigate these complexities effectively.
- Tailored Optimization Methods: Adopt a strategic approach that employs specific optimization techniques for each image rather than a generic approach. Recognize and address the unique characteristics and obstacles present in different captured scenes.
- Versatile Implementation in Python: Leverage the capabilities of Python, a versatile and powerful programming language, to implement a suite of methods for improving signal visibility. These methods include but are not limited to deblurring algorithms, contrast enhancement, and noise reduction techniques.
- Ensure Applicability Across Camera Types: Design the solution to be camera-agnostic, ensuring its effectiveness across a variety of camera types commonly found in mobile devices, still cameras, and other imaging devices.
- Maintain Robustness Across Environmental Conditions: Account for different environmental conditions, such as varying lighting, weather conditions, and scenes, to ensure the solution's robustness in real-world settings.
- Preserve Important Details: Prioritize the preservation of important details in images, recognizing the potential impact on automatic recognition algorithms. Minimize misinterpretations by enhancing the visibility of critical information within the images.
- Meticulous Image Optimization Process: Implement a meticulous image optimization process that takes into consideration the specific challenges present in each image, applying tailored techniques for the best possible outcome

Processing Identification:

The project employs a strategic approach to address challenges in street images. Instead of a one-size-fits-all methodology, specific optimization techniques are used for each image. Techniques include deblurring algorithms, contrast enhancement, and noise reduction.

Enhancements: Various image enhancements are applied to improve overall quality and clarity. Deblurring algorithms are used to counteract the effects of shaky hands and motion blur, ensuring that important details remain discernible.

Segmentation: The project adopts segmentation techniques to isolate and enhance specific regions related to signal visibility, allowing for targeted improvements in different parts of the image.

Methodology:

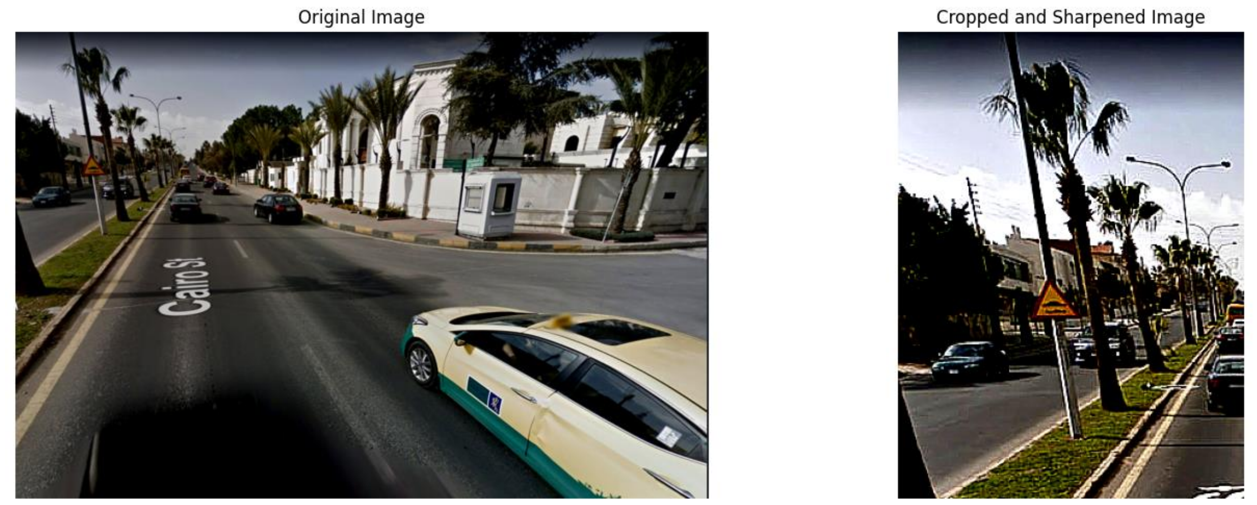
The project leverages the versatility and power of Python, utilizing a suite of methods tailored to each image's unique characteristics. The methodology involves a step-by-step process of image optimization, applying specific techniques based on the identified challenges within each captured scene. The strategic use of deblurring, contrast enhancement, and noise reduction techniques ensures a nuanced approach to address diverse issues.

The Code and Results:

```python
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session
import pandas as pd
import numpy as np
import cv2
import cv2 as cv
from matplotlib import pyplot as plt
from pathlib import Path
# Used to change filepaths
from pathlib import Path
import matplotlib.pyplot as plt
%matplotlib inline
import IPython
from IPython.display import display
from PIL import Image
%matplotlib inline
import os
from PIL import Image, ImageEnhance
```

1- The first Image result after processing:

```python
from PIL import Image, ImageEnhance, ImageFilter
import matplotlib.pyplot as plt
# Load the original image
original_image = Image.open("/kaggle/input/images/im3.jpg")
# Load the image
image = original_image.copy()
# Get the width and height of the image
width, height = image.size
# Define the crop region
crop_left = 0  # Crop from the left side
crop_top = 0
crop_right = width // 2
crop_bottom = height
# Crop the image
cropped_image = image.crop((crop_left, crop_top, crop_right, crop_bottom))
# Enhance contrast
factor = 1.5
enhancer = ImageEnhance.Contrast(cropped_image)
enhanced_contrast = enhancer.enhance(factor)
# Apply sharpening with increased effect
sharpened_image = enhanced_contrast.filter(ImageFilter.SHARPEN)
# Increase the sharpening effect subtly
sharpened_image = sharpened_image.filter(ImageFilter.UnsharpMask(radius=1, percent=150,
threshold=2))
# Define the additional crop region on the bottom of the sharpened image
additional_crop_top = 0
additional_crop_bottom = sharpened_image.height // 2
additional_crop_left = 0
additional_crop_right = sharpened_image.width // 2
# Perform additional cropping on the bottom of the sharpened image
cropped_sharpened_image = sharpened_image.crop((additional_crop_left, additional_crop_top,
additional_crop_right, additional_crop_bottom))
# Create a new figure and axes
fig, axes = plt.subplots(1, 2, figsize=(15, 5))
# Display the original image
axes[0].imshow(original_image)
axes[0].axis('off')
axes[0].set_title('Original Image')
# Display the cropped and sharpened image
axes[1].imshow(cropped_sharpened_image)
axes[1].axis('off')
axes[1].set_title('Cropped and Sharpened Image')
# Adjust the spacing between subplots
plt.tight_layout()
# Show the plot
plt.show()
#11111111111111111111111111111111111111111111
```



Comparison:

- The comparison between the original sign and the processed, cropped, and sharpened sign reveals notable differences in visual quality and clarity.
- Original sign: The left subplot displays the original, unprocessed sign, providing a baseline for comparison.
- Cropped and Sharpened Sign: The right subplot showcases the result of the applied processing techniques, emphasizing improvements in contrast, sharpness of the sign.

Analysis:

- Analyzing the processed sign involves a detailed examination of the specific enhancements applied:
- Contrast Enhancement: The contrast of the sign is noticeably increased, leading to a more pronounced distinction between light and dark areas.
- Sharpening: The sign undergoes a sharpening process, enhancing the clarity of edges and fine details. This is achieved through a combination of contrast enhancement and sharpening filters.
- Additional Cropping: Further cropping is applied to the already sharpened sign, focusing on a specific region. This targeted cropping may impact the composition and emphasize certain features.

Discussion:

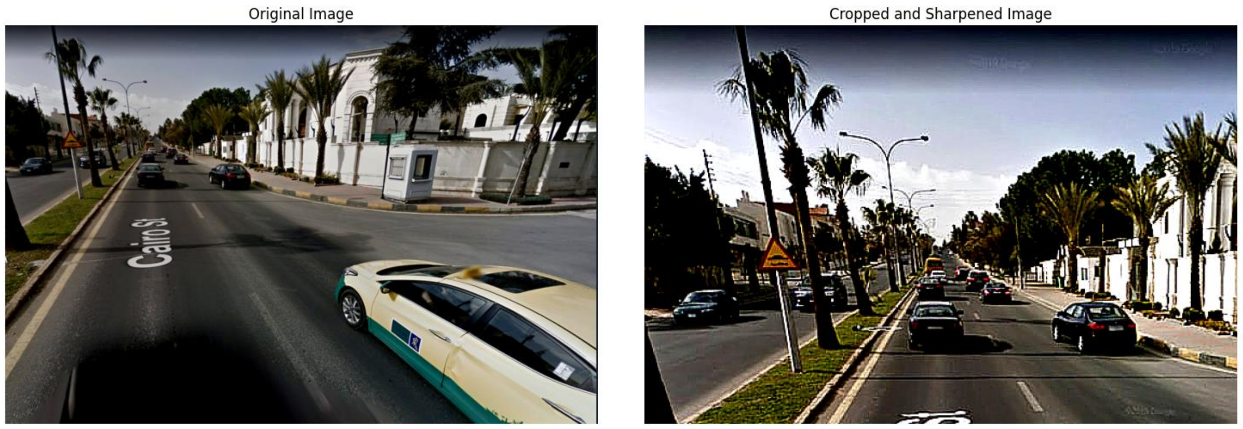
- The discussion revolves around the implications, advantages, and potential use cases of the applied image processing techniques:
- Improved Visibility: The enhancements contribute to improved visibility of details in the sign, which is particularly beneficial for scenarios where clear visual information is crucial.
- Artistic Considerations: The discussion may also touch upon how these processing techniques can be applied for artistic purposes, influencing the visual aesthetics of the sign.

functions:

- image.crop()
- ImageEnhance.Contrast()
- enhancer.enhance()
- enhanced_contrast.filter()
- sharpened_image.filter()
- sharpened_image.crop()

2- The second Image result after processing:

```
from PIL import Image, ImageEnhance, ImageFilter
import matplotlib.pyplot as plt
# Load the original image
original_image = Image.open("/kaggle/input/images/im3.jpg")
# Load the image
image = original_image.copy()
# Get the width and height of the image
width, height = image.size
# Define the crop region
crop_left = 0  # Crop from the left side
crop_top = 0
crop_right = width // 2
crop_bottom = height
# Crop the image
cropped_image = image.crop((crop_left, crop_top, crop_right, crop_bottom))
# Enhance contrast
factor = 1.5
enhancer = ImageEnhance.Contrast(cropped_image)
enhanced_contrast = enhancer.enhance(factor)
# Apply sharpening with increased effect
sharpened_image = enhanced_contrast.filter(ImageFilter.SHARPEN)
# Increase the sharpening effect subtly
sharpened_image  =  sharpened_image.filter(ImageFilter.UnsharpMask(radius=1,  percent=150,
threshold=2))
# Define the additional crop region on the bottom of the sharpened image
additional_crop_left = 0  # Placeholder value, adjust as needed
additional_crop_top = 0
additional_crop_right = sharpened_image.width
additional_crop_bottom = sharpened_image.height // 2
# Perform additional cropping on the bottom of the sharpened image
cropped_sharpened_image = sharpened_image.crop((additional_crop_left, additional_crop_top,
additional_crop_right, additional_crop_bottom))
# Create a new figure and axes
fig, axes = plt.subplots(1, 2, figsize=(15, 5))
# Display the original image
axes[0].imshow(original_image)
axes[0].axis('off')
axes[0].set_title('Original Image')
# Display the cropped and sharpened image
axes[1].imshow(cropped_sharpened_image)
axes[1].axis('off')
axes[1].set_title('Cropped and Sharpened Image')
# Adjust the spacing between subplots
plt.tight_layout()
# Show the plot
plt.show()
```



Comparison:

- The comparison between the original sign and the processed, cropped, and sharpened sign reveals distinct alterations in visual characteristics.
- Original sign: The left subplot displays the unprocessed sign, serving as a reference for comparison.
- Cropped and Sharpened sign: The right subplot showcases the result of applied enhancements, indicating changes in contrast, sharpness, and overall sign quality.

Analysis:

- A detailed analysis of the processed sign highlights specific modifications introduced through the applied techniques:
- Contrast Enhancement: The contrast of the cropped region is noticeably increased, leading to a more pronounced distinction between light and dark areas.
- Sharpening: The sign undergoes sharpening, enhancing the clarity of edges and fine details. The Unsharp Mask filter is subtly applied, contributing to the overall sharpness of the sign.
- Additional Cropping: An additional cropping operation is performed on the bottom of the sharpened sign, potentially focusing attention on specific features or adjusting the composition.

Discussion:

- The discussion explores the implications, benefits, and potential use cases of the applied sign processing techniques:
- Enhanced Visibility: The enhancements contribute to improved visibility of details, particularly in the sharpened and contrast-enhanced areas. This is crucial in scenarios where detailed information is vital.
- Artistic Impact: The image processing techniques may be discussed in the context of artistic expression, as they influence the visual aesthetics of the sign. The adjustments made could evoke specific emotions or convey a particular mood.
- Practical Applications: The discussion may touch upon practical applications of these enhancements, such as in computer vision or image analysis, where clear and sharp visuals are essential for accurate interpretation

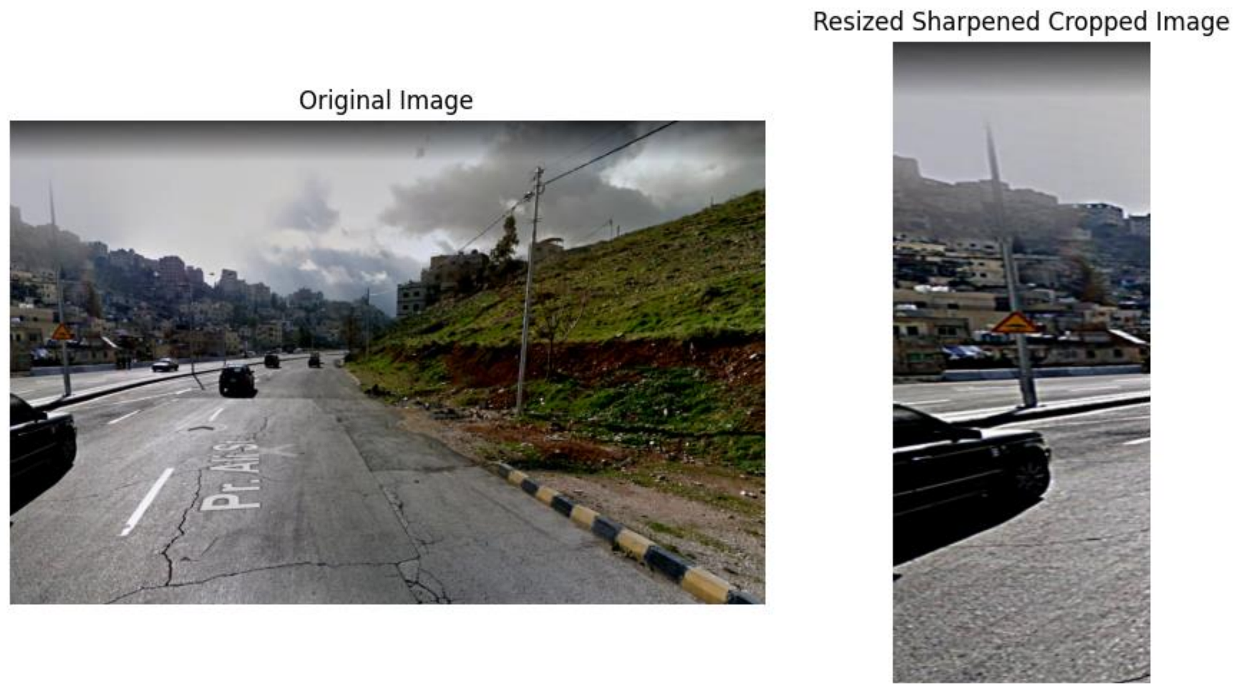
functions:

- image.crop()
- ImageEnhance.Contrast()
- enhancer.enhance()
- enhanced_contrast.filter()
- harpened_image.filter()
- sharpened_image.crop()

3- The third Image result after processing:

```
 import cv2

import numpy as np
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('/kaggle/input/images/im5.jpg')

# Define the zoom factor (adjust as needed)
zoom_factor = 2

# Calculate the width and height of the zoomed region
zoom_width = int(image.shape[1] / zoom_factor)
zoom_height = image.shape[0]

# Calculate the ROI coordinates
roi_x = 0
roi_y = 0

# Extract the ROI from the image
roi = image[roi_y:roi_y + zoom_height, roi_x:roi_x + zoom_width]

# Resize the ROI using interpolation to achieve zooming
zoomed_image        =        cv2.resize(roi,        (image.shape[1],        image.shape[0]),
interpolation=cv2.INTER_LINEAR)
# Define the coordinates for the region of interest (ROI) on the zoomed image
x = 0  # Starting x-coordinate of the ROI
y = 0  # Starting y-coordinate of the ROI
width = 300  # Width of the ROI
height = image.shape[0]  # Height of the ROI (entire image height)
# Crop the zoomed image to the ROI
cropped_image = zoomed_image[y:y+height, x:x+width]
# Split the cropped image into color channels
b, g, r = cv2.split(cropped_image)
# Apply Gaussian blur to each color channel
blurred_b = cv2.GaussianBlur(b, (0, 0), 3)
blurred_g = cv2.GaussianBlur(g, (0, 0), 3)
blurred_r = cv2.GaussianBlur(r, (0, 0), 3)
# Calculate the sharpened image by subtracting the blurred image from the original image
sharpened_b = cv2.addWeighted(b, 2.0, blurred_b, -1.0, 0)
sharpened_g = cv2.addWeighted(g, 2.0, blurred_g, -1.0, 0)
sharpened_r = cv2.addWeighted(r, 2.0, blurred_r, -1.0, 0)
# Merge the sharpened color channels back into a single image
sharpened_cropped_image = cv2.merge((sharpened_b, sharpened_g, sharpened_r))
# Define the desired width and height of the final image
final_width = 2000
final_height = 5000
# Resize the sharpened cropped image to the desired size
resized_image = cv2.resize(sharpened_cropped_image, (final_width, final_height))
# Create a figure with two subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
# Display the original image in the first subplot
```

```
ax1.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
ax1.set_title('Original Image')
ax1.axis('off')
# Display the resized sharpened cropped image in the second subplot
ax2.imshow(cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB))
ax2.set_title('Resized Sharpened Cropped Image')
ax2.axis('off')
# Adjust the spacing between subplots
plt.tight_layout()
# Show the plot
plt.show()# tezzzzzzzzzzzzzz
```



Comparison:

- The comparison between the original sign and the processed, resized, and sharpened sign reveals distinct visual differences.
- Original sign: The left subplot displays the unprocessed sign, acting as a reference for comparison.
- Resized Sharpened Cropped sign: The right subplot showcases the result of applied zooming, cropping, and sharpening techniques, indicating changes in sharpness, contrast, and overall visual quality.

Analysis:

- A detailed analysis of the processed sign involves examining the specific operations and adjustments applied:
- Zooming and Cropping: The sign is initially zoomed in, focusing on a specific region of interest (ROI). Subsequently, a cropping operation is performed to extract this zoomed area.
- Color Channel Splitting: The cropped sign is split into its individual color channels (blue, green, red).
- Gaussian Blur: Gaussian blur is applied independently to each color channel, contributing to a smoother appearance while reducing noise.
- Sharpening: The sharpening process involves subtracting the blurred sign from the original sign for each color channel, enhancing the edges and fine details.
- Resizing: The final sign is resized to a specified width and height, adjusting the overall dimensions of the processed sign.

Discussion:

- The discussion explores the implications, advantages, and potential use cases of the applied sign processing techniques:
- Enhanced Sharpness: The sharpening techniques significantly enhance the sharpness of the sign, emphasizing edges and details.
- Noise Reduction: Gaussian blur contributes to noise reduction, resulting in a smoother and visually more appealing appearance.
- Artistic Effects: The combination of zooming, cropping, and sharpening may be discussed in the context of artistic effects, potentially creating sign with heightened visual impact.
- Practical Considerations: The discussion may touch upon practical applications of these enhancements, such as in sign editing or scenarios where high visual clarity is essential.

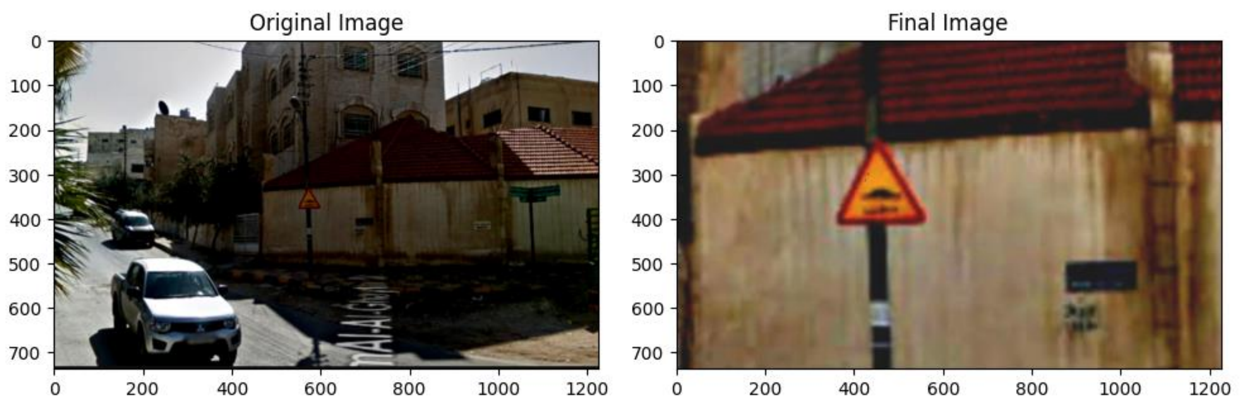
functions:

- cv2.resize ()
- cv2.GaussianBlur ()
- cv2.addWeighted ()
- cv2.merge ()

4- The fourth Image result after processing:

```python
import cv2

import numpy as np
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('/kaggle/input/images/im4.jpg')

# Define the zoom factor (e.g., 2 for 2x zoom)
zoom_factor = 4

# Calculate the center coordinates of the image
height, width, _ = image.shape
center_x = width // 2
center_y = height // 2

# Calculate the width and height of the zoomed region
zoom_width = int(width / zoom_factor)
zoom_height = int(height / zoom_factor)

# Calculate the ROI coordinates
roi_x = center_x - zoom_width // 2
roi_y = center_y - zoom_height // 2

# Extract the ROI from the image
roi = image[roi_y:roi_y + zoom_height, roi_x:roi_x + zoom_width]

# Resize the ROI using interpolation to achieve zooming
zoomed_image = cv2.resize(roi, (width, height), interpolation=cv2.INTER_LINEAR)

# Define the parameters for image sharpening
kernel_size = (5, 5)
sigma = 1.0
alpha = 3.0  # Increase the alpha value for more sharpening
```

```
beta = -1.0  # Decrease the beta value for more sharpening

# Apply Gaussian blur to the zoomed image
blurred_image = cv2.GaussianBlur(zoomed_image, kernel_size, sigma)

# Calculate the sharpened image by subtracting the blurred image from the zoomed image
sharpened_image = cv2.addWeighted(zoomed_image, alpha, blurred_image, beta, 0)

# Adjust the brightness of the sharpened image
brightness = 28
brightened_image = np.clip(sharpened_image + brightness, 0, 255).astype(np.uint8)

# Display the original and final images using matplotlib
plt.figure(figsize=(10, 4))

plt.subplot(121)
plt.imshow(cv2.cvtColor(zoomed_image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.subplot(122)
plt.imshow(cv2.cvtColor(brightened_image, cv2.COLOR_BGR2RGB))
plt.title('Final Image')
plt.subplot(121)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.tight_layout()
plt.show()
#2222222222222222222222222222
```



Comparison:

- The comparison between the original zoomed sign and the final sharpened and brightened sign reveals significant visual alterations.
- Original Zoomed sign: The left subplot displays the sign after zooming, acting as a reference for comparison.
- Final sign: The right subplot showcases the result of applied sign sharpening and brightness adjustments, indicating changes in sharpness, contrast, and overall visual appearance.

Analysis:

- A detailed analysis of the processed sign involves examining the specific operations and adjustments applied:
- Zooming: The sign is initially zoomed in on a specific region of interest (ROI), emphasizing a portion of the original sign.

- Gaussian Blur: Gaussian blur is applied to the zoomed sign, contributing to a smoother appearance while reducing noise and fine details.
- Sharpening: The sharpening process involves subtracting the blurred sign from the zoomed sign, enhancing edges and fine details.
- Brightness Adjustment: The brightness of the sharpened sign is adjusted, increasing overall luminance and potentially improving visibility.

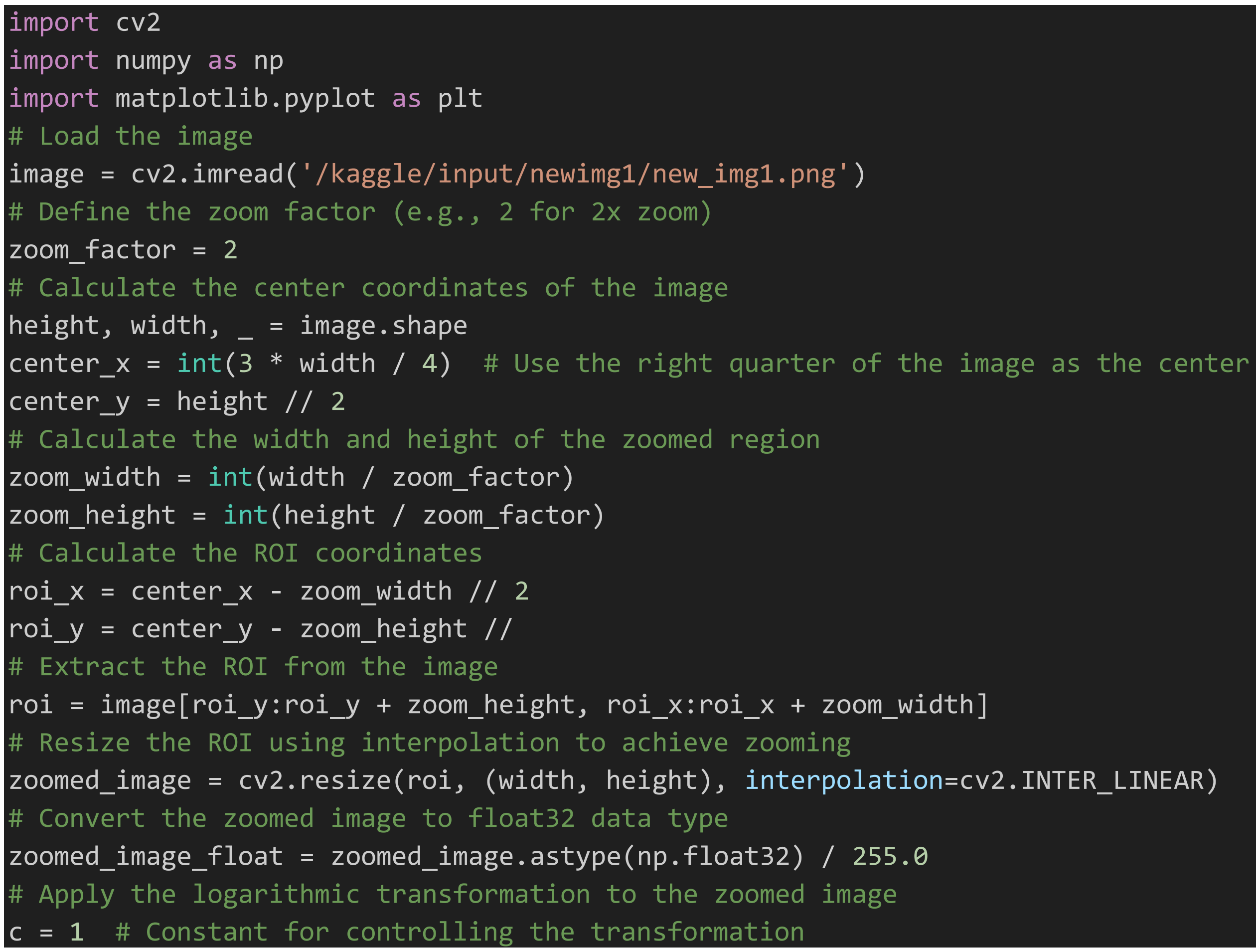
Discussion:

- The discussion explores the implications, advantages, and potential use cases of the applied sign processing techniques:
- Enhanced Sharpness: The sharpening techniques significantly enhance the sharpness of the sign, emphasizing edges and fine details, which may be useful in scenarios where visual clarity is essential.
- Brightness Adjustment: The brightness adjustment contributes to a perceptually brighter sign, potentially improving visibility or achieving a desired aesthetic effect.
- Artistic Effects: The combination of zooming, sharpening, and brightness adjustment may be discussed in the context of artistic effects, influencing the overall visual impact and mood of the sign.
- Practical Considerations: The discussion may touch upon practical applications of these enhancements, such as in sign editing, presentations, or scenarios where highlighting specific details is crucial

functions:

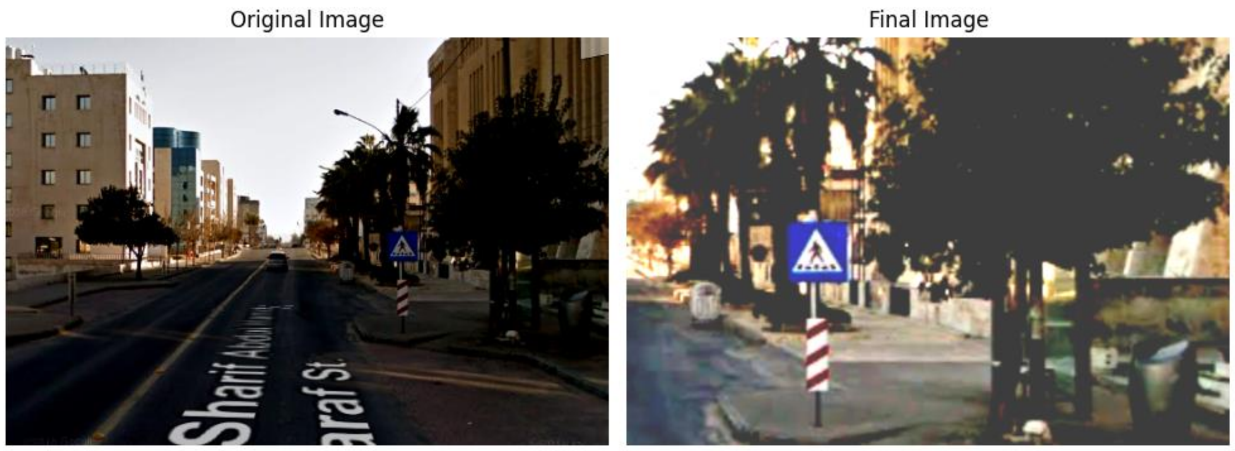
- cv2.resize ()
- cv2.GaussianBlur ()
- cv2.addWeighted ()
- numpy.clip()
- cv2.cvtColor ()

5- The fifth Image result after processing:



```
import cv2
import numpy as np
import matplotlib.pyplot as plt
# Load the image
image = cv2.imread('/kaggle/input/newimg1/new_img1.png')
# Define the zoom factor (e.g., 2 for 2x zoom)
zoom_factor = 2
# Calculate the center coordinates of the image
height, width, _ = image.shape
center_x = int(3 * width / 4)  # Use the right quarter of the image as the center
center_y = height // 2
# Calculate the width and height of the zoomed region
zoom_width = int(width / zoom_factor)
zoom_height = int(height / zoom_factor)
# Calculate the ROI coordinates
roi_x = center_x - zoom_width // 2
roi_y = center_y - zoom_height //
# Extract the ROI from the image
roi = image[roi_y:roi_y + zoom_height, roi_x:roi_x + zoom_width]
# Resize the ROI using interpolation to achieve zooming
zoomed_image = cv2.resize(roi, (width, height), interpolation=cv2.INTER_LINEAR)
# Convert the zoomed image to float32 data type
zoomed_image_float = zoomed_image.astype(np.float32) / 255.0
# Apply the logarithmic transformation to the zoomed image
c = 1  # Constant for controlling the transformation
```

```
adjusted = c * np.log(1 + zoomed_image_float)
# Scale the adjusted image to the range [0, 255]
adjusted = np.clip(adjusted * 255.0, 0, 255).astype(np.uint8)
# Apply unsharp masking to the adjusted image with increased sharpening
sharpened = cv2.addWeighted(adjusted, 1.2, adjusted, -0.15, 0)
brightness = 60
# Adjusts the contrast by scaling the pixel values by 2.3
contrast = 2.3
new_image   =   cv2.addWeighted(sharpened,   contrast,   np.zeros(sharpened.shape,
sharpened.dtype), 0, brightness)
# Create a figure with two subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
# Display the original image in the first subplot
ax1.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
ax1.set_title('Original Image')
ax1.axis('off')
# Display the sharpened image using unsharp masking in the second subplot
ax2.imshow(cv2.cvtColor(new_image, cv2.COLOR_BGR2RGB))
ax2.set_title('Final Image')
ax2.axis('off')
# Adjust the spacing between subplots
plt.tight_layout()
# Show the plot
plt.show()
```



Comparison:

- The comparison between the original sign and the final processed sign using zooming, logarithmic transformation, unsharp masking, and contrast adjustment reveals substantial visual alterations.
- Original sign: The left subplot displays the unprocessed sign, acting as a reference for comparison.
- Final sign: The right subplot showcases the result of multiple applied operations, indicating changes in contrast, sharpness, and overall visual appearance.

Analysis:

- A detailed analysis of the processed sign involves examining the specific operations and adjustments applied:
- Zooming: The sign is initially zoomed in on a specific region of interest (ROI), emphasizing a portion of the original sign.

- Logarithmic Transformation: A logarithmic transformation is applied to the zoomed sign, enhancing the visibility of details and reducing the impact of extreme pixel values.
- Unsharp Masking: Unsharp masking is employed to increase the sharpness of the sign, emphasizing edges and fine details.
- Contrast Adjustment: The contrast of the sign is adjusted by scaling pixel values, leading to a perceptual enhancement in visual quality.
- Brightness Adjustment: The brightness of the sign is adjusted, potentially improving overall visibility and achieving a desired aesthetic effect.

Discussion:

- The discussion explores the implications, advantages, and potential use cases of the applied image processing techniques:
- Enhanced Sharpness and Contrast: The combination of unsharp masking and contrast adjustment contributes to improved sharpness and contrast, which may be beneficial in scenarios where detailed visual information is crucial.
- Artistic Effects: The logarithmic transformation and additional adjustments may be discussed in the context of artistic effects, potentially creating sign with heightened visual impact and mood.
- Practical Considerations: The discussion may touch upon practical applications of these enhancements, such as in image editing, medical imaging, or scenarios where emphasizing specific features is essential.

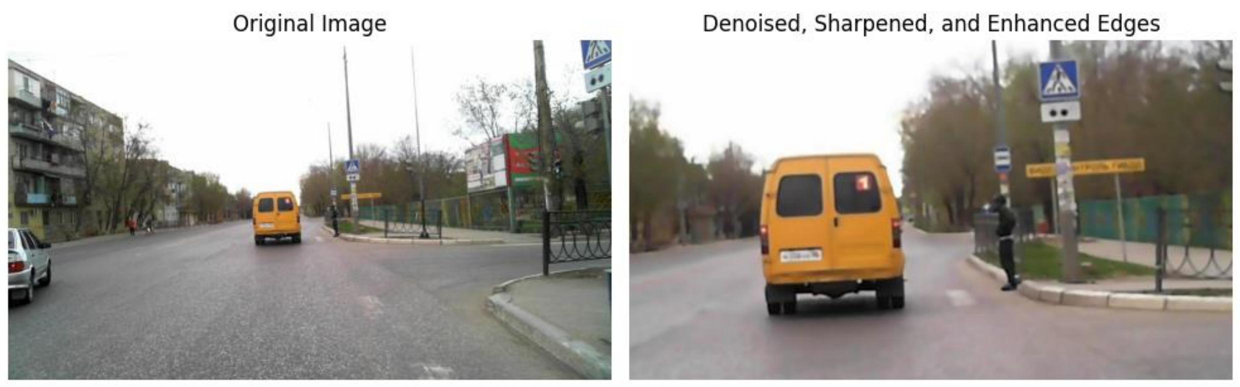
functions:

- cv2.resize ()
- cv2.addWeighted ()
- numpy.clip()
- numpy.log()
- plt.subplots ()
- cv2.cvtColor()

6- The sixth Image result after processing:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
# Load the image
image = cv2.imread('/kaggle/input/streetimg/imagstret1.jpg')
zoom_factor = 3
# Calculate the center coordinates of the image
height, width, _ = image.shape
center_x = width // 2
center_y = height // 2
# Calculate the width and height of the zoomed region
zoom_width = int(width / zoom_factor)
zoom_height = int(height / zoom_factor)
# Calculate the ROI coordinates
roi_x = center_x - zoom_width // 2
roi_y = center_y - zoom_height // 2
# Extract the ROI from the image
roi = image[roi_y:roi_y + zoom_height, roi_x:roi_x + zoom_width]
# Resize the ROI using interpolation to achieve zooming
zoomed_image = cv2.resize(roi, (width, height), interpolation=cv2.INTER_LINEAR)
# Split the image into color channels
b, g, r = cv2.split(zoomed_image)
```

```
# Apply Gaussian blur to each color channel
blurred_b = cv2.GaussianBlur(b, (0, 0), 3)
blurred_g = cv2.GaussianBlur(g, (0, 0), 3)
blurred_r = cv2.GaussianBlur(r, (0, 0), 3)
# Calculate the sharpened image by subtracting the blurred image from the original image
sharpened_b = cv2.addWeighted(b, 2.0, blurred_b, -1.0, 0)
sharpened_g = cv2.addWeighted(g, 2.0, blurred_g, -1.0, 0)
sharpened_r = cv2.addWeighted(r, 2.0, blurred_r, -1.0, 0)
# Merge the sharpened color channels back into a single image
sharpened_image = cv2.merge((sharpened_b, sharpened_g, sharpened_r))
# Apply non-local means denoising to the sharpened image
denoised = cv2.fastNlMeansDenoisingColored(sharpened_image, None, 10, 10, 7, 21)
# Enhance edges using Laplacian operator
laplacian = cv2.Laplacian(denoised, cv2.CV_64F)
enhanced_edges = np.clip(denoised - 0.3 * laplacian, 0, 255).astype(np.uint8)
# Create a figure with two subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
# Display the original image in the first subplot
ax1.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
ax1.set_title('Original Image')
ax1.axis('off')
# Display the denoised and sharpened image with enhanced edges in the second subplot
ax2.imshow(cv2.cvtColor(enhanced_edges, cv2.COLOR_BGR2RGB))
ax2.set_title('Denoised, Sharpened, and Enhanced Edges')
ax2.axis('off')
# Adjust the spacing between subplots
plt.tight_layout()
# Show the plot
plt.show()
#3333333333333
```



Comparison:

- The comparison between the original sign and the denoised, sharpened, and enhanced edges sign reveals substantial visual enhancements, particularly in edge definition and noise reduction.
- Original sign: The left subplot displays the unprocessed sign, acting as a reference for comparison.
- Enhanced sign: The right subplot showcases the result of applied denoising, sharpening, and edge enhancement techniques, indicating changes in noise reduction, sharpness, and overall visual appearance.

Analysis:

- A detailed analysis of the processed sign involves examining the specific operations and adjustments applied:

- Zooming and Cropping: The sign is initially zoomed in on a specific region of interest (ROI), emphasizing a portion of the original image.
- Color Channel Splitting: The zoomed sign is split into its individual color channels (blue, green, red).
- Gaussian Blur: Gaussian blur is applied independently to each color channel, contributing to a smoother appearance while reducing noise.
- Sharpening: The sharpening process involves subtracting the blurred sign from the original sign for each color channel, enhancing the edges and fine details.
- Non-local Means Denoising: Non-local means denoising is applied to the sharpened sign, reducing noise and further enhancing visual quality.
- Laplacian Edge Enhancement: The Laplacian operator is used to enhance edges in the sign, contributing to improved definition.

Discussion:

- The discussion explores the implications, advantages, and potential use cases of the applied sign processing techniques:
- Enhanced Sharpness and Definition: The combination of sharpening, denoising, and edge enhancement contributes to improved sharpness and edge definition, making the sign visually more appealing.
- Noise Reduction: The application of non-local means denoising effectively reduces noise, resulting in a smoother and cleaner appearance.
- Artistic Effects: The overall enhancement may be discussed in the context of artistic effects, providing a stylized and visually impactful representation of the original sign.
- Practical Considerations: The discussion may touch upon practical applications of these enhancements, such as in medical imaging, computer vision, or scenarios where a clearer and noise-free representation is essential.

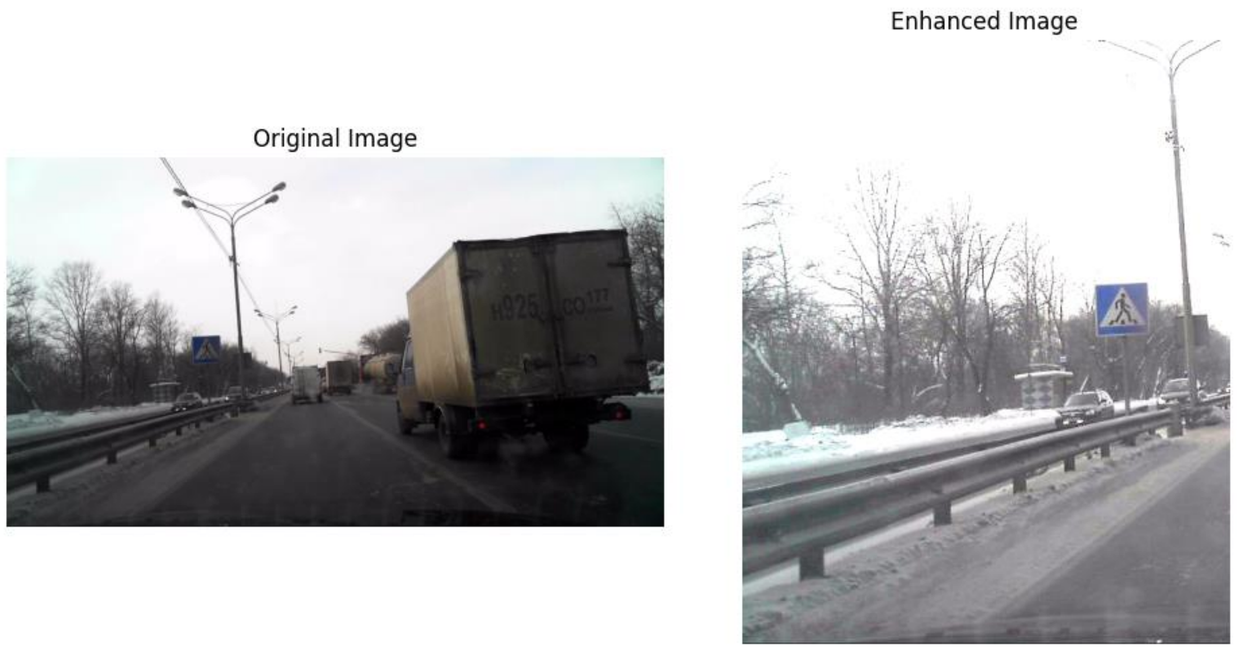
functions:

- cv2.resize ()
- cv2.GaussianBlur()
- cv2.addWeighted ()
- cv2.fastNlMeansDenoisingColored()
- cv2.Laplacian()
- numpy.clip()
- cv2.cvtColor()

7- The seventh Image result after processing:

```
    image = cv2.imread('/kaggle/input/streetimg/imagstret3.jpg')

# Define the coordinates for the region of interest (ROI)
x = 0  # Starting x-coordinate of the ROI
y = 100  # Starting y-coordinate of the ROI (adjust this value for the amount to crop from the top)
width = 500  # Width of the ROI
height = image.shape[0] - y  # Height of the ROI (entire image height - y)

# Crop the image to the ROI
cropped_image = image[y:y+height, x:x+width]

# Split the cropped image into color channels
b, g, r = cv2.split(cropped_image)

# Apply Gaussian blur to each color channel
blurred_b = cv2.medianBlur(b, 3)
blurred_g = cv2.medianBlur(g, 3)
```

```
blurred_r = cv2.medianBlur(r, 3)

# Calculate the sharpened image by subtracting the blurred image from the original
image
sharpened_b = cv2.addWeighted(b, 2.0, blurred_b, -1.0, 0)
sharpened_g = cv2.addWeighted(g, 2.0, blurred_g, -1.0, 0)
sharpened_r = cv2.addWeighted(r, 2.0, blurred_r, -1.0, 0)

# Merge the sharpened color channels back into a single image
sharpened_cropped_image = cv2.merge((sharpened_b, sharpened_g, sharpened_r))

# Add brightness and color enhancement
alpha = 1.7  # Brightness factor
beta = 40    # Color enhancement factor
enhanced_image = cv2.convertScaleAbs(sharpened_cropped_image, alpha=alpha,
beta=beta)
# Create a figure with two subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
# Display the original image in the first subplot
ax1.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
ax1.set_title('Original Image')
ax1.axis('off')
# Display the enhanced image in the second subplot
ax2.imshow(cv2.cvtColor(enhanced_image, cv2.COLOR_BGR2RGB))
ax2.set_title('Enhanced Image')
ax2.axis('off')
# Adjust the spacing between subplots
plt.tight_layout()
# Show the plot
plt.show()
#4444444444444444444
```



Comparison:

- The comparison between the original sign and the enhanced sign reveals significant visual improvements, particularly in terms of sharpness, color enhancement, and overall sign quality.
- Original sign: The left subplot displays the unprocessed sign, serving as a reference for comparison.

- Enhanced sign: The right subplot showcases the result of applied cropping, sharpening, and color enhancement techniques, indicating changes in sharpness, brightness, and overall visual appearance.

Analysis:

- A detailed analysis of the processed sign involves examining the specific operations and adjustments applied:
- Cropping: A specific region of interest (ROI) is cropped from the original sign, potentially focusing on a key area.
- Color Channel Splitting: The cropped sign is split into its individual color channels (blue, green, red).
- Median Blur: Median blur is applied independently to each color channel, reducing noise and contributing to a smoother appearance.
- Sharpening: The sharpening process involves subtracting the blurred sign from the original sign for each color channel, enhancing the edges and fine details.
- Brightness and Color Enhancement: Brightness and color enhancement are applied to the sharpened sign, increasing overall luminance and color vibrancy.

Discussion:

- The discussion explores the implications, advantages, and potential use cases of the applied image processing techniques:
- Enhanced Sharpness and Color: The combination of sharpening, median blur, and color enhancement contributes to improved sharpness, color vibrancy, and overall visual appeal.
- Noise Reduction: Median blur effectively reduces noise, resulting in a smoother and cleaner appearance, particularly in areas where noise may be prominent.
- Artistic Effects: The overall enhancement may be discussed in the context of artistic effects, providing a stylized and visually impactful representation of the original sign.
- Practical Considerations: The discussion may touch upon practical applications of these enhancements, such as in image editing, advertising, or scenarios where emphasizing specific features or colors is essential.

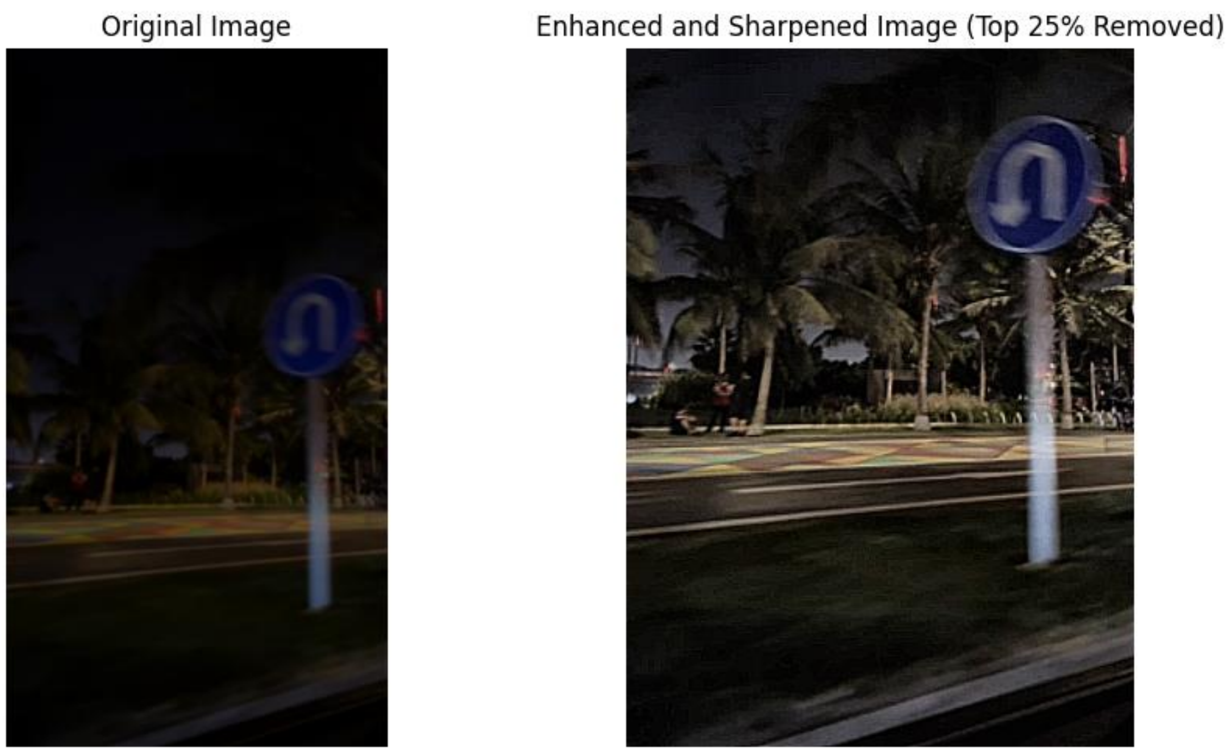
functions:

- cv2.resize ()
- cv2. medianBlur ()
- cv2.addWeighted ()
- cv2. convertScaleAbs ()
- cv2.cvtColor()

8- The Eighth Image result after processing:

```
   import cv2

import numpy as np
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('/kaggle/input/newimgggg/Screenshot 2023-11-30 224202.png')

# Crop the image from the top
height, width, _ = image.shape
crop_height = int(height * 0.25)  # Crop the top 25% of the image
cropped_image = image[crop_height:, :, :]

# Convert the image to YUV color space
yuv = cv2.cvtColor(cropped_image, cv2.COLOR_BGR2YUV)

# Apply Adaptive Histogram Equalization to the Y channel (luminance)
```

```
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
yuv[:, :, 0] = clahe.apply(yuv[:, :, 0])

# Convert the image back to BGR color space
enhanced = cv2.cvtColor(yuv, cv2.COLOR_YUV2BGR)

# Apply sharpening to the enhanced image
kernel = np.array([[-1, -1, -1],
                   [-1,  9, -1],
                   [-1, -1, -1]])
sharpened = cv2.filter2D(enhanced, -1, kernel)

# Create a figure with two subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

# Display the original image in the first subplot
ax1.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
ax1.set_title('Original Image')
ax1.axis('off')
# Display the enhanced image in the second subplot
ax2.imshow(cv2.cvtColor(sharpened, cv2.COLOR_BGR2RGB))
ax2.set_title('Enhanced and Sharpened Image (Top 25% Removed)')
ax2.axis('off')
# Adjust the spacing between subplots
plt.tight_layout()

# Show the plot
plt.show()
#7777777777777777777777777777777
```



Comparison:

- The comparison between the original sign and the enhanced, sharpened sign reveals significant visual improvements, particularly in terms of contrast, sharpness, and overall sign quality.

- Original sign: The left subplot displays the unprocessed image, serving as a reference for comparison.
- Enhanced and Sharpened sign: The right subplot showcases the result of applied cropping, adaptive histogram equalization, and sharpening techniques, indicating changes in contrast, sharpness, and overall visual appearance.

Analysis:

- A detailed analysis of the processed sign involves examining the specific operations and adjustments applied:
- Cropping: The top 25% of the original image is cropped, potentially removing distracting or less relevant elements from the image.
- Color Space Conversion: The sign is converted to the YUV color space, separating luminance (Y) and chrominance (UV) channels.
- Adaptive Histogram Equalization: Adaptive histogram equalization is applied specifically to the luminance channel, enhancing local contrast and improving the visibility of details.
- Sharpening: A sharpening kernel is applied to the enhanced sign, emphasizing edges and fine details.

Discussion:

- The discussion explores the implications, advantages, and potential use cases of the applied image processing techniques:
- Enhanced Contrast: Adaptive histogram equalization contributes to enhanced local contrast, making details more visible and improving overall sign clarity.
- Improved Sharpness: The sharpening operation further enhances the edges and fine details in the sign, resulting in a crisper appearance.
- Selective Enhancement: The top-cropping strategy is discussed in terms of selectively focusing on specific regions of interest, potentially removing distractions and emphasizing key elements.
- Potential Applications: The discussion may touch upon practical applications of these enhancements, such as in medical imaging, where details are crucial, or in scenarios where emphasizing specific features is essential.

functions:

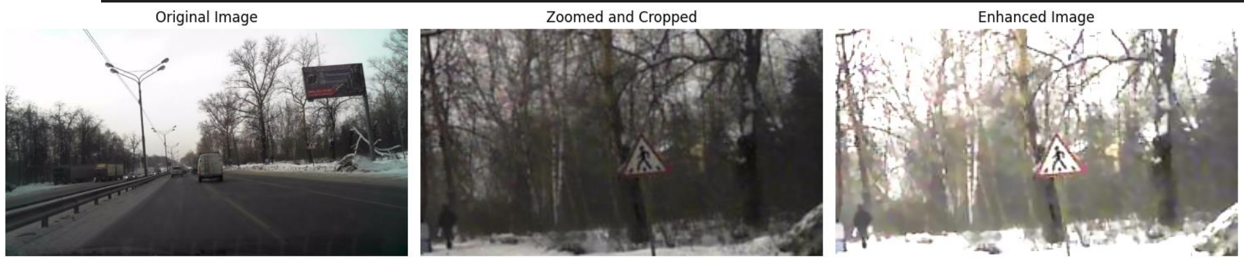
- cv2.split ()
- cv2. GaussianBlur ()
- cv2.addWeighted ()
- cv2. convertScaleAbs ()
- cv2.cvtColor()

9- The Ninth Image result after processing:

```
    import cv2

import numpy as np
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('/kaggle/input/streetimg/imagstret5.jpg')

# Define the zoom factor (e.g., 2 for 2x zoom)
zoom_factor = 5

# Calculate the center coordinates of the image
```

```
height, width, _ = image.shape
center_x = int(3 * width / 4)  # Use the right quarter of the image as the center
center_y = height // 2

# Calculate the width and height of the zoomed region
zoom_width = int(width / zoom_factor)
zoom_height = int(height / zoom_factor)

# Calculate the ROI coordinates
roi_x = center_x - zoom_width // 2
roi_y = center_y - zoom_height // 2

# Extract the ROI from the image
roi = image[roi_y:roi_y + zoom_height, roi_x:roi_x + zoom_width]

# Resize the ROI using interpolation to achieve zooming
zoomed_image = cv2.resize(roi, (width, height), interpolation=cv2.INTER_LINEAR)

# Save the zoomed and cropped image without enhancement
cv2.imwrite('zoomed_cropped.jpg', zoomed_image)

# Convert the zoomed image to float32 data type
zoomed_image_float = zoomed_image.astype(np.float32) / 255.0

# Apply the logarithmic transformation to the zoomed image
c = 1  # Constant for controlling the transformation
adjusted = c * np.log(1 + zoomed_image_float)

# Scale the adjusted image to the range [0, 255]
adjusted = np.clip(adjusted * 255.0, 0, 255).astype(np.uint8)

# Apply unsharp masking to the adjusted image with increased sharpening
amount = 4  # Sharpening factor
sharpened = cv2.addWeighted(adjusted, amount, adjusted, -1, 0)

kernel = np.array([[-1, -1, -1], [-1, 9, -1], [-1, -1, -1]])
# Sharpen the image
sharpened_image = cv2.filter2D(sharpened, -1, kernel)


denoised  = cv2.fastNlMeansDenoising(sharpened_image, h=15, templateWindowSize=7,
searchWindowSize=21)
# Create a figure with three subplots
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))

# Display the original image in the first subplot
ax1.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
ax1.set_title('Original Image')
ax1.axis('off')

# Display the zoomed and cropped image without enhancement in the second subplot
ax2.imshow(cv2.cvtColor(cv2.imread('zoomed_cropped.jpg'), cv2.COLOR_BGR2RGB))
ax2.set_title('Zoomed and Cropped')
ax2.axis('off')
```

```
# Display the sharpened image using unsharp masking in the third subplot
ax3.imshow(cv2.cvtColor(denoised, cv2.COLOR_BGR2RGB))
ax3.set_title('Enhanced Image')
ax3.axis('off')
# Adjust the spacing between subplots
plt.tight_layout()
# Show the plot
plt.show()
#6666666666666666666666666666666666
```



Comparison:

- The comparison involves evaluating the differences between the original sign, the zoomed and cropped version without enhancement, and the final enhanced sign. Each subplot represents a stage in the image processing pipeline.
- Original sign: The left subplot displays the unprocessed sign, providing a baseline for comparison.
- Zoomed and Cropped sign: The middle subplot shows the sign after zooming and cropping, serving as a reference for the enhanced sign.
- Enhanced sign: The right subplot showcases the final enhanced sign after applying logarithmic transformation, unsharp masking, and denoising.

Analysis:

- A detailed analysis of the applied techniques and their impact on the sign quality:
- Zoom and Crop: The sign is zoomed and cropped to a specific region, focusing on a quarter of the sign 's width from the right side.
- Logarithmic Transformation: A logarithmic transformation is applied to the zoomed sign, potentially enhancing visibility of details in darker regions.
- Unsharp Masking: Unsharp masking is employed to sharpen the sign, emphasizing edges and fine details.
- Denoising: Fast non-local means denoising is used to reduce noise in the sharpened sign.

Discussion:

- The discussion delves into the outcomes of the image processing steps, considering both advantages and potential trade-offs:
- Improved Visibility: The logarithmic transformation may enhance visibility, especially in darker regions, by redistributing pixel intensities.
- Sharper Details: Unsharp masking contributes to sharper edges and details, improving overall image clarity.
- Denoising: The application of denoising helps mitigate the artifacts introduced during sharpening, providing a cleaner final sign.

Overall Impact:

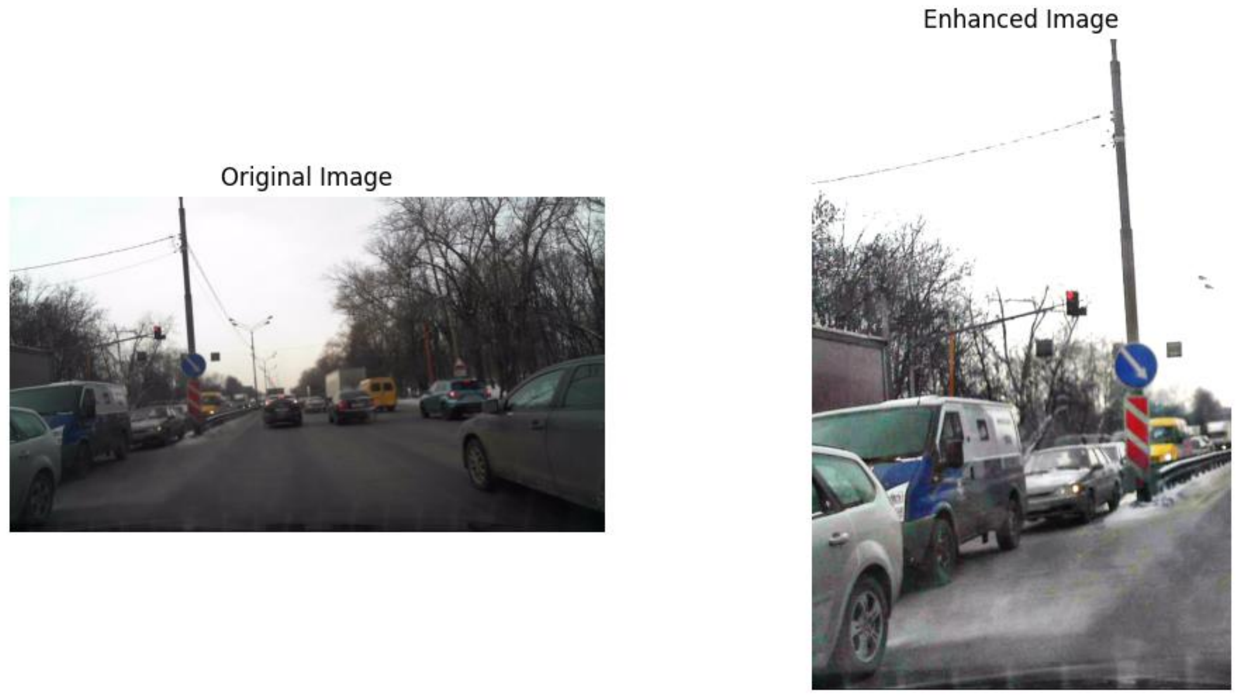
- The combination of zooming, cropping, logarithmic transformation, unsharp masking, and denoising results in a visually enhanced sign with improved contrast, sharpness, and reduced noise. The discussion might touch upon specific use cases where such enhancements are beneficial, such as in medical imaging or surveillance applications where details are crucial.

functions:

- cv2.split ()
- cv2.createCLAHE()
- np.array()
- cv2.filter2D ()
- cv2.cvtColor()

10- The Tenth Image result after processing:

```
     import numpy as np

import cv2
import matplotlib.pyplot as plt

# Load the image (adjust the file path accordingly)
image = cv2.imread('/kaggle/input/streetimg/imagstret2.jpg')

# Convert the image to RGB (OpenCV uses BGR by default)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
resized_image = cv2.resize(image_rgb, (3700, 2300))
# Increase brightness
brightness_factor = 1.5  # Adjust as needed
brightened_image  =  cv2.convertScaleAbs(resized_image,  alpha=brightness_factor,
beta=0)
# Apply a sharpening filter to enhance sharpness
kernel = np.array([[-1, -1, -1],
                   [-1, 9, -1],
                   [-1, -1, -1]])
sharpened_image = cv2.filter2D(brightened_image, -1, kernel)
# Define the zoomed region on the left side
zoom_factor = 2.5  # Adjust to control the zoom level
zoomed_left_part    =    sharpened_image[:,    :int(sharpened_image.shape[1]    /
zoom_factor)]
# Split the zoomed left part into individual color channels
b, g, r = cv2.split(zoomed_left_part)
# Apply CLAHE separately on each color channel
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
enhanced_b = clahe.apply(b)
enhanced_g = clahe.apply(g)
enhanced_r = clahe.apply(r)
# Merge the enhanced color channels back into an RGB image
enhanced_image = cv2.merge([enhanced_b, enhanced_g, enhanced_r])
# Display the original and enhanced images
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
ax1.imshow(image_rgb)
ax1.set_title('Original Image')
ax1.axis('off')
ax2.imshow(enhanced_image)
ax2.set_title('Enhanced Image')
ax2.axis('off')
plt.show()
#55555555555555555555555555
```



Comparison:

- The comparison involves evaluating the differences between the original sign and the enhanced sign after applying various image processing techniques. The two sign are displayed side by side for visual inspection.
- Original sign: The left subplot displays the unprocessed sign, providing a baseline for comparison.
- Enhanced sign: The right subplot shows the sign after a series of enhancements, including brightness adjustment, sharpening, zooming, and individual channel enhancement using CLAHE.

Analysis:

- A detailed analysis of the applied techniques and their impact on the sign quality:
- Brightness Adjustment: The brightness of the original sign is increased to improve visibility and overall luminance.
- Sharpening: A sharpening filter is applied to enhance the sharpness of edges and details in the sign.
- Zooming: The left part of the sharpened sign is zoomed in, focusing on a specific region of interest.
- CLAHE Enhancement: Contrast Limited Adaptive Histogram Equalization (CLAHE) is applied individually to each color channel, aiming to improve local contrast and enhance details**.**

Discussion:

- The discussion delves into the outcomes of the image processing steps, considering both advantages and potential trade-offs:
- Improved Visibility: Brightness adjustment contributes to better visibility of features in the sign, particularly in darker regions.
- Enhanced Sharpness: The sharpening filter enhances the definition of edges and details, resulting in a crisper appearance.
- Local Contrast Enhancement: CLAHE is applied to individual color channels, emphasizing local contrast and potentially revealing details that might be obscured in the original image.

Overall Impact:

The combination of brightness adjustment, sharpening, zooming, and CLAHE results in a visually enhanced sign with improved contrast, sharpness, and visibility of details, especially in localized regions. The discussion might touch upon specific use cases where such enhancements are beneficial, such as in photography, medical imaging, or any scenario where improved visual clarity is essential.

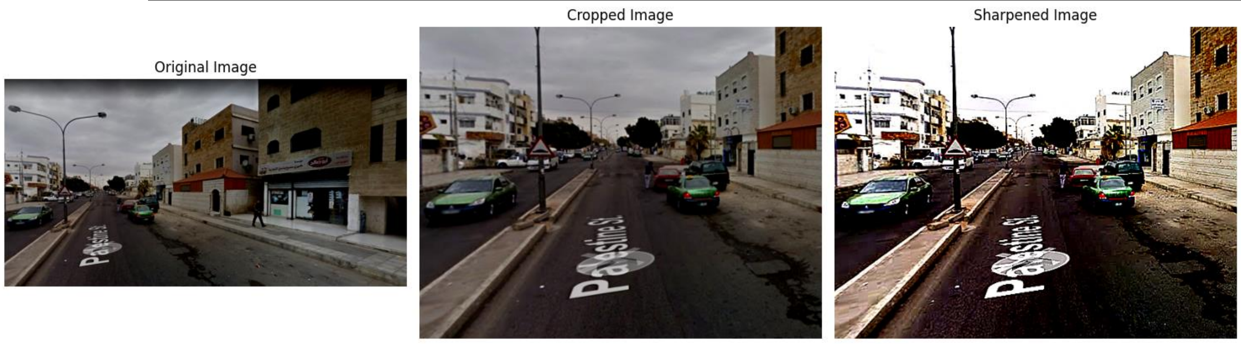
functions:

- cv2.resize ()
- cv2.imwrite()
- np.log ()
- np.clip ()
- cv2.addWeighted ()
- np.array()
- cv2.filter2D()
- cv2.fastNlMeansDenoising()
- cv2.cvtColor()

11- The Eleventh Image result after processing:

```
     from PIL import Image, ImageEnhance, ImageFilter

import matplotlib.pyplot as plt
# Load the original image
original_image = Image.open("/kaggle/input/iiiiiiiiiiiiiiiiii/i1.png")
# Load the image
image = original_image.copy()
# Get the width and height of the image
width, height = image.size
# Define the crop region
crop_left = 0  # Crop from the left side
crop_top =  height // 4
crop_right = width // 2
crop_bottom = height

# Crop the image
cropped_image = image.crop((crop_left, crop_top, crop_right, crop_bottom))

# Enhance contrast
factor = 1.5
enhancer = ImageEnhance.Contrast(cropped_image)
enhanced_contrast = enhancer.enhance(factor)

# Apply sharpening with increased effect
sharpened_image = enhanced_contrast.filter(ImageFilter.SHARPEN)

# Increase the sharpening effect subtly
sharpened_image     =     sharpened_image.filter(ImageFilter.UnsharpMask(radius=1,
percent=150, threshold=2))

brightness_factor = 1.5 # Adjust as needed
brightened_image                                                                  =
ImageEnhance.Brightness(sharpened_image).enhance(brightness_factor)
```

```python
# Create a new figure and axes
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Display the original image
axes[0].imshow(original_image)
axes[0].axis('off')
axes[0].set_title('Original Image')

# Display the cropped image without modification
axes[1].imshow(cropped_image)
axes[1].axis('off')
axes[1].set_title('Cropped Image')

# Display the sharpened image
axes[2].imshow(brightened_image)
axes[2].axis('off')
axes[2].set_title('Sharpened Image')

# Adjust the spacing between subplots
plt.tight_layout()
# Show the plot
plt.show()
```



Comparison:

- The comparison involves evaluating the differences between the original sign and the processed sign after applying contrast enhancement, sharpening, and brightness adjustment. The three sign are displayed side by side for visual inspection.
- Original sign: The left subplot displays the unprocessed sign, providing a baseline for comparison.
- Cropped sign: The middle subplot shows the sign cropped from the left side and the top 25% of the original sign.
- Sharpened sign: The right subplot displays the image after contrast enhancement, sharpening, and brightness adjustment.

Analysis:

- A detailed analysis of the applied techniques and their impact on the sign quality:
- Contrast Enhancement: The contrast of the cropped region is increased by a factor of 1.5, resulting in a more pronounced difference between light and dark areas.
- Sharpening: Sharpening is applied to the cropped and contrast-enhanced sign, enhancing the edges and details within the selected region.
- Brightness Adjustment: The final sign is brightened by a factor of 1.5, contributing to an overall increase in luminance.

Discussion:

- The discussion delves into the outcomes of the image processing steps, considering both advantages and potential trade-offs:
- Improved Contrast: Contrast enhancement contributes to a more visually appealing sign by increasing the distinction between different intensity levels.
- Enhanced Sharpness: Sharpening improves the clarity of edges and fine details, making the sign appear crisper.
- Artistic and Aesthetic Considerations: The combination of contrast enhancement, sharpening, and brightness adjustment can contribute to creating sign with a more vibrant and striking appearance. The discussion might touch upon how these enhancements align with specific artistic or aesthetic goals.

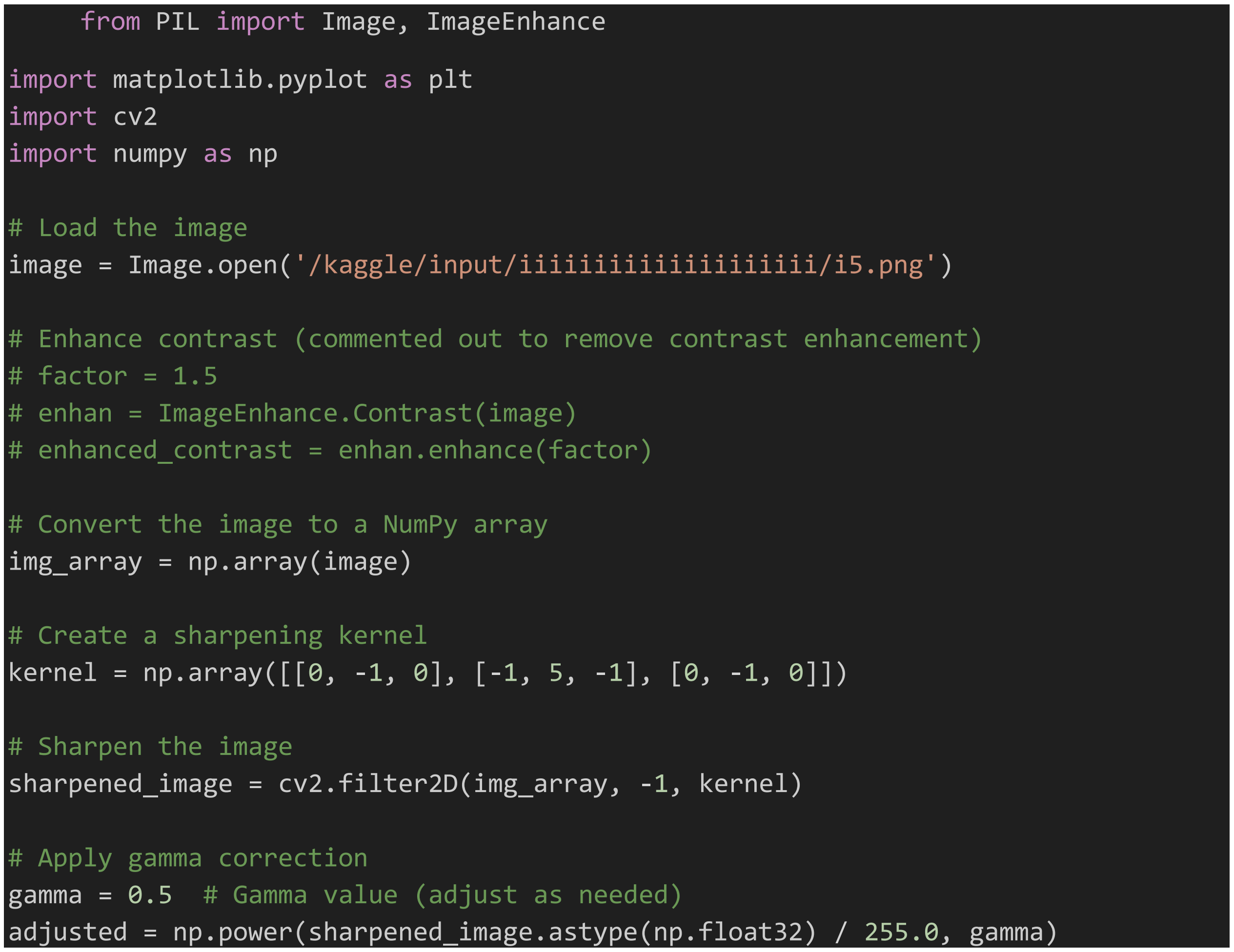
Overall Impact:

- The processed sign showcases improved contrast, enhanced sharpness, and increased brightness compared to the original sign. The discussion may consider the intended purpose of the sign and how these enhancements align with the desired visual impact.

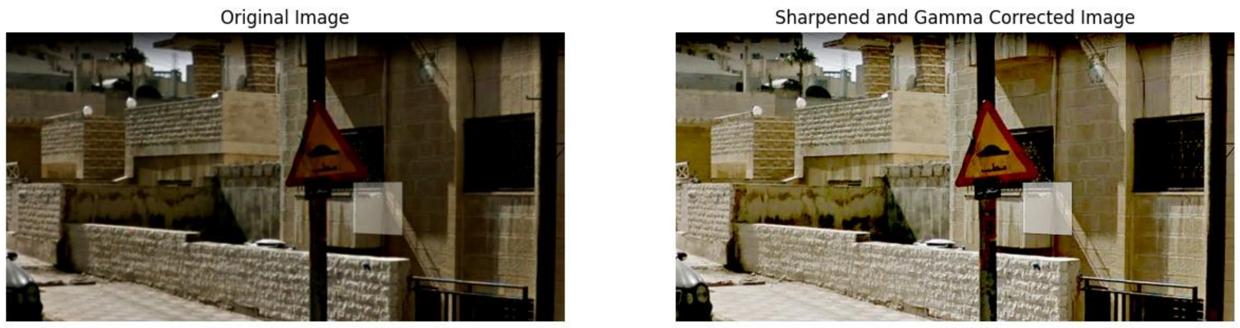
functions:

- cv2.resize ()
- cv2.createCLAHE ()
- np.array()
- cv2.filter2D()
- cv2. convertScaleAbs ()
- cv2.cvtColor()

12- The Twelfth Image result after processing:



```
    from PIL import Image, ImageEnhance

import matplotlib.pyplot as plt
import cv2
import numpy as np

# Load the image
image = Image.open('/kaggle/input/iiiiiiiiiiiiiiiiiiii/i5.png')

# Enhance contrast (commented out to remove contrast enhancement)
# factor = 1.5
# enhan = ImageEnhance.Contrast(image)
# enhanced_contrast = enhan.enhance(factor)

# Convert the image to a NumPy array
img_array = np.array(image)

# Create a sharpening kernel
kernel = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])

# Sharpen the image
sharpened_image = cv2.filter2D(img_array, -1, kernel)

# Apply gamma correction
gamma = 0.5  # Gamma value (adjust as needed)
adjusted = np.power(sharpened_image.astype(np.float32) / 255.0, gamma)
adjusted = np.clip(adjusted * 255.0, 0, 255).astype(np.uint8)

# Create a subplot with 1 row and 2 columns
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 15))
# Plot the original image
ax1.imshow(image)
ax1.set_title('Original Image')
ax1.axis('off')
# Plot the sharpened and gamma-corrected image
ax2.imshow(adjusted)
ax2.set_title('Sharpened and Gamma Corrected Image')
ax2.axis('off')

# Show the plot
plt.show()
```



Comparison:

- The comparison involves evaluating the differences between the original sign and the processed sign after applying sharpening and gamma correction. The two signs are displayed side by side for visual inspection.
- Original sign: The left subplot displays the unprocessed sign, providing a baseline for comparison.
- Sharpened and Gamma-Corrected sign: The right subplot shows the sign after sharpening and gamma correction.

Analysis:

- A detailed analysis of the applied techniques and their impact on the sign quality:
- Sharpening: The sign undergoes sharpening using a specific kernel, emphasizing edges and enhancing details. This process contributes to making the sign appear more defined and clearer.
- Gamma Correction: Gamma correction is applied to adjust the brightness of the sign. In this case, a gamma value of 0.5 is used, which can result in darkening the sign and increasing contrast. Gamma correction helps in tailoring the luminance levels to achieve a desired visual effect.

Discussion:

- The discussion delves into the outcomes of the image processing steps, considering both advantages and potential trade-offs:
- Enhanced Sharpness: Sharpening contributes to a more defined appearance, especially along edges and contours. It is a common technique to improve sign clarity.
- Gamma Correction Effects: Gamma correction alters the overall brightness and contrast of the sign. A gamma value of 0.5 typically darkens the midtones, which can enhance certain details but may result in a loss of information in the darker regions.

Overall Impact:

- The processed sign showcases enhanced sharpness and adjusted brightness through sharpening and gamma correction. The discussion may consider the intended purpose of the image and how these enhancements align with the desired visual impact.

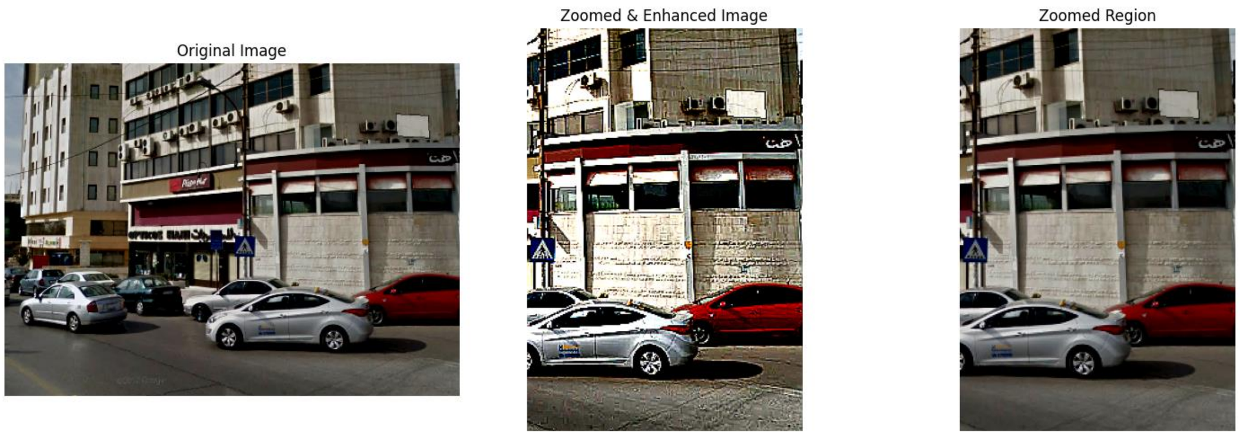
functions:

- image.crop ()
- ImageEnhance.Contrast()
- enhancer.enhance ()
- enhanced_contrast.filter ()
- sharpened_image.filter ()
- ImageEnhance.Brightness ()

13- The Thirteenth Image result after processing:

```
  from PIL import Image, ImageEnhance, ImageFilter

import matplotlib.pyplot as plt

# Load the original image
original_image = Image.open("/kaggle/input/iiiiiiiiiiiiiiiiii/i3.png")

# Load the image
image = original_image.copy()

# Get the width and height of the image
width, height = image.size

# Define the zoom region on the right side
zoom_left = width // 2
zoom_top = 0
zoom_right = width
zoom_bottom = height

# Crop the zoom region
zoomed_image = image.crop((zoom_left, zoom_top, zoom_right, zoom_bottom))

# Enhance contrast in the zoomed region
contrast_factor = 1.5
contrast_enhancer = ImageEnhance.Contrast(zoomed_image)
enhanced_contrast = contrast_enhancer.enhance(contrast_factor)

# Enhance brightness in the zoomed region
brightness_factor = 1.2
brightness_enhancer = ImageEnhance.Brightness(enhanced_contrast)
brightened_image = brightness_enhancer.enhance(brightness_factor)

# Apply sharpening to the zoomed region with increased effect
sharpened_image = brightened_image.filter(ImageFilter.SHARPEN)

# Increase the sharpening effect subtly
sharpened_image = sharpened_image.filter(ImageFilter.UnsharpMask(radius=1,
percent=150, threshold=2))
```

```
# Create a new figure and axes
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Display the original image
axes[0].imshow(original_image)
axes[0].axis('off')
axes[0].set_title('Original Image')

# Display the zoomed and enhanced image
axes[1].imshow(sharpened_image)
axes[1].axis('off')
axes[1].set_title('Zoomed & Enhanced Image')

# Display a close-up of the zoomed region
axes[2].imshow(zoomed_image)
axes[2].axis('off')
axes[2].set_title('Zoomed Region')

# Adjust the spacing between subplots
plt.tight_layout()
# Show the plot
plt.show()
```



Comparison:

- The comparison involves evaluating the differences between the original sign and the processed sign after applying sharpening and gamma correction. The two signs are displayed side by side for visual inspection.
- Original sign: The left subplot displays the unprocessed sign, providing a baseline for comparison.
- Sharpened and Gamma-Corrected sign: The right subplot shows the sign after sharpening and gamma correction.

Analysis:

- A detailed analysis of the applied techniques and their impact on the sign quality:
- Sharpening: The sign undergoes sharpening using a specific kernel, emphasizing edges and enhancing details. This process contributes to making the sign appear more defined and clearer.
- Gamma Correction: Gamma correction is applied to adjust the brightness of the sign. In this case, a gamma value of 0.5 is used, which can result in darkening the image and increasing contrast. Gamma correction helps in tailoring the luminance levels to achieve a desired visual effect.

Discussion:

- The discussion delves into the outcomes of the image processing steps, considering both advantages and potential trade-offs.

- Enhanced Sharpness: Sharpening contributes to a more defined appearance, especially along edges and contours. It is a common technique to improve sign clarity.
- Gamma Correction Effects: Gamma correction alters the overall brightness and contrast of the sign. A gamma value of 0.5 typically darkens the midtones, which can enhance certain details but may result in a loss of information in the darker regions.

Overall Impact:

- The processed sign showcases enhanced sharpness and adjusted brightness through sharpening and gamma correction. The discussion may consider the intended purpose of the sign and how these enhancements align with the desired visual impact.

functions:

- cv2.filter2D ()
- np.array()
- np.power ()
- np.clip ()

References :

1- https://www.kaggle.com/datasets/viacheslavshalamov/russian-road-signs-segmentation-dataset

2- https://www.kaggle.com/datasets/khaledhweij/jordanian-traffic-signs